

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

average 1 hour per response, including the time for reviewing instructions, searching existing data sources gathering and maintaining the data regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Information and Regulatory Affairs, Office of

AD-A234 834

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED

Final: Nov 30, 1990 to Mar 1, 1993

4. TITLE AND SUBTITLE

Ada Compiler Validation Summary Report: DDC International A/S, DACS Sun3/SunOS Native Ada Compiler System, Version 4.6, Sun3/60 (Host & Target), 901129S1.11076

5. FUNDING NUMBERS

6. AUTHOR(S)

National Institute of Standards and Technology
Gaithersburg, MD
USA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Institute of Standards and Technology
National Computer Systems Laboratory
Bldg. 255, Rm A266
Gaithersburg, MD 20899 USA

8. PERFORMING ORGANIZATION
REPORT NUMBER

NIST90DDC500_5_1.11

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Ada Joint Program Office
United States Department of Defense
Pentagon, RM 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY
REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

DDC International A/S, DACS Sun3/SunOS Native Ada Compiler System, Version 4.6, Gaithersburg, MD, Sun3/60 running SunOS, Version 4.0_Export (Host & Target), ACVC 1.11.

14. SUBJECT TERMS

Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION
OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION
UNCLASSIFIED

19. SECURITY CLASSIFICATION
OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

AVF Control Number: NIST90DDC500_5_1.11
DATE COMPLETED
BEFORE ON-SITE: October 30, 1990
AFTER ON-SITE: November 30, 1990
REVISIONS:

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901129S1.11076
DDC International A/S
DACS Sun3/SunOS Native Ada Compiler System, Version 4.6
Sun3/60 => Sun3/60

Prepared By:
Software Standards Validation Group
National Computer Systems Laboratory
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, Maryland 20899

AVF Control Number: NIST90DDC500_5_1.11

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on November 29, 1990.

Compiler Name and Version: DACS Sun3/SunOS Native Ada Compiler System, Version 4.6

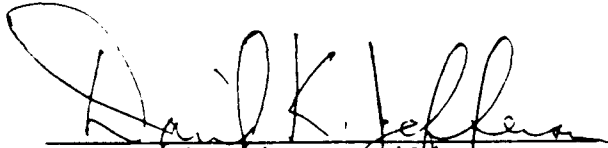
Host Computer System: Sun3/60 running SunOS, Version 4.0_Export

Target Computer System: Sun3/60 running SunOS, Version 4.0_Export

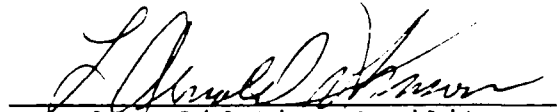
A more detailed description of this Ada implementation is found in section 3.1 of this report.

As a result of this validation effort, Validation Certificate 901129S1.11076 is awarded to DDC International A/S. This certificate expires on March 01, 1993.

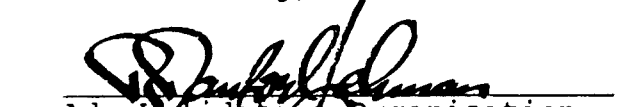
This report has been reviewed and is approved.



Ada Validation Facility
Dr. David K. Jefferson
Chief, Information Systems
Engineering Division (ISED)
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



Ada Validation Facility
Mr. L. Arnold Johnson
Manager, Software Standards
Validation Group
National Computer Systems
Laboratory (NCSL)
National Institute of
Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899



Ada Validation Organization
Director, Computer & Software
Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

The following declaration of conformance was supplied by the customer.

DECLARATION OF CONFORMANCE

Customer and Certificate Awardee: DDC International A/S

Ada Validation Facility: National Institute of Standards and
Technology
National Computer Systems Laboratory
(NCSL)
Software Validation Group
Building 225, Room A266
Gaithersburg, Maryland 20899

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: DACS Sun3/SunOS Native Ada Compiler
System, Version 4.6

Host Computer System: Sun3/60 running SunOS, Version
4.0_Export

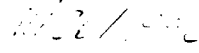
Target Computer System: Sun3/60 running SunOS, Version
4.0_Export

Declaration:

[I/we] the undersigned, declare that [I/we] have no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A ISO 8652-1987 in the implementation listed above.



Customer Signature
Company
Title



Date

TABLE OF CONTENTS

CHAPTER 1	1-1
INTRODUCTION	1-1
1.1 USE OF THIS VALIDATION SUMMARY REPORT	1-1
1.2 REFERENCES	1-1
1.3 ACVC TEST CLASSES	1-2
1.4 DEFINITION OF TERMS	1-3
CHAPTER 2	2-1
IMPLEMENTATION DEPENDENCIES	2-1
2.1 WITHDRAWN TESTS	2-1
2.2 INAPPLICABLE TESTS	2-1
2.3 TEST MODIFICATIONS	2-4
CHAPTER 3	3-1
PROCESSING INFORMATION	3-1
3.1 TESTING ENVIRONMENT	3-1
3.2 SUMMARY OF TEST RESULTS	3-2
3.3 TEST EXECUTION	3-2
APPENDIX A	A-1
MACRO PARAMETERS	A-1
APPENDIX B	B-1
COMPILATION SYSTEM OPTIONS	B-1
LINKER OPTIONS	B-2
APPENDIX C	C-1
APPENDIX F OF THE Ada STANDARD	C-1

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued. Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3. For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing

withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 3.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, Validation consisting of the test suite, the support programs, the ACVC Capability user's guide and the template for the validation summary (ACVC) report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.
Conformity	Fulfillment by a product, process or service of all requirements specified.

Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

Some tests are withdrawn by the AVO from the ACVC because they do not conform to the Ada Standard. The following 81 tests had been withdrawn by the Ada Validation Organization (AVO) at the time of validation testing. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 90-10-12.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026A
B83026B	C83041A	B85001L	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. The inapplicability criteria for some tests are explained in documents issued by ISO and the AJPO known as Ada Issues and commonly referenced in the format AI-dddd. For this implementation, the following tests were inapplicable for the reasons indicated; references to Ada Issues are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)

C45641L..Y (14 tests)

C46012L..Z (15 tests)

C24113I..K (3 TESTS) USE A LINE LENGTH IN THE INPUT FILE WHICH EXCEEDS 126 CHARACTERS.

C35702A, C35713B, C45423B, B86001T, AND C86006H CHECK FOR THE PREDEFINED TYPE SHORT_FLOAT.

C35713D AND B86001Z CHECK FOR A PREDEFINED FLOATING-POINT TYPE WITH A NAME OTHER THAN FLOAT, LONG_FLOAT, OR SHORT_FLOAT.

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35404D, C45231D, B86001X, C86006E, and CD7101G check for a predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

C45531M, C45531N, C45531O, C45531P, C45532M, C45532N, C45532O, AND C45532P USE FINE 48 BIT FIXED POINT BASE TYPES WHICH ARE NOT SUPPORTED BY THIS COMPILER.

C45624A CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOW IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 5. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOW IS TRUE.

C45624B CHECKS THAT THE PROPER EXCEPTION IS RAISED IF MACHINE_OVERFLOW IS FALSE FOR FLOATING POINT TYPES WITH DIGITS 6. FOR THIS IMPLEMENTATION, MACHINE_OVERFLOW IS TRUE.

C4A013B CONTAINS THE EVALUATION OF AN EXPRESSION INVOLVING 'MACHINE_RADIX APPLIED TO THE MOST PRECISE FLOATING-POINT TYPE. THIS EXPRESSION WOULD RAISE AN EXCEPTION. SINCE THE EXPRESSION MUST BE STATIC, IT IS REJECTED AT COMPILE TIME.

C86001F RECOMPILES PACKAGE SYSTEM, MAKING PACKAGE TEXT_IO, AND HENCE PACKAGE REPORT, OBSOLETE. FOR THIS IMPLEMENTATION, THE PACKAGE TEXT_IO IS DEPENDENT UPON PACKAGE SYSTEM.

B86001Y CHECKS FOR A PREDEFINED FIXED-POINT TYPE OTHER THAN DURATION.

C96005B CHECKS FOR VALUES OF TYPE DURATION'BASE THAT ARE OUTSIDE THE RANGE OF DURATION. THERE ARE NO SUCH VALUES FOR THIS IMPLEMENTATION.

CA2009C, CA2009F, BC3204C, AND BC3205D THESE TESTS INSTANTIATE GENERIC UNITS BEFORE THEIR BODIES ARE COMPILED. THIS IMPLEMENTATION CREATES A DEPENDENCE ON GENERIC UNIT AS ALLOWED BY AI-00408 & AI-00530 SUCH THAT AT THE COMPILATION OF THE GENERIC UNIT BODIES MAKES THE INSTANTIATING UNITS OBSOLETE.

CD1009C USES A REPRESENTATION CLAUSE SPECIFYING A NON-DEFAULT SIZE FOR A FLOATING-POINT TYPE.

CD2A84A, CD2A84E, CD2A84I..J (2 TESTS), AND CD2A84O USE REPRESENTATION CLAUSES SPECIFYING NON-DEFAULT SIZES FOR ACCESS TYPES.

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	-----	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

THE TESTS LISTED IN THE FOLLOWING TABLE ARE NOT APPLICABLE BECAUSE THE GIVEN FILE OPERATIONS ARE NOT SUPPORTED FOR THE GIVEN COMBINATION OF MODE AND FILE ACCESS METHOD.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO

CE2203A CHECKS FOR SEQUENTIAL_IO THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS

IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

EE2401D CHECKS WHETHER READ, WRITE, SET_INDEX, INDEX, SIZE, AND END_OF_FILE ARE SUPPORTED FOR DIRECT FILES FOR AN UNCONSTRAINED ARRAY TYPE. USE_ERROR WAS RAISED FOR DIRECT CREATE. THE MAXIMUM ELEMENT SIZE SUPPORTED FOR DIRECT_IO IS 32K BYTES.

CE2403A CHECKS FOR DIRECT_IO THAT WRITE RAISES USE_ERROR IF THE CAPACITY OF THE EXTERNAL FILE IS EXCEEDED. THIS IMPLEMENTATION CANNOT RESTRICT FILE CAPACITY.

CE3111B AND CE3115A SIMULTANEOUSLY ASSOCIATE INPUT AND OUTPUT FILES WITH THE SAME EXTERNAL FILE, AND EXPECT THAT OUTPUT IS IMMEDIATELY WRITTEN TO THE EXTERNAL FILE AND AVAILABLE FOR READING; THIS IMPLEMENTATION BUFFERS FILES, AND EACH TEST'S ATTEMPT TO READ SUCH OUTPUT (AT LINES 87 & 101, RESPECTIVELY) RAISES END_ERROR.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B CHECKS THAT PAGE RAISES LAYOUT_ERROR WHEN THE VALUE OF THE PAGE NUMBER EXCEEDS COUNT'LAST. THE VALUE OF COUNT'LAST IS GREATER THAN 150000 AND THE CHECKING OF THIS OBJECTIVE IS IMPRACTICAL.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 65 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B22003A	B26001A	B26002A	B26005A	B28003A	B29001A	B33301B
B35101A	B37106A	B37301B	B37302A	B38003A	B38003B	B38009A
B38009B	B55A01A	B61001C	B61001F	B61001H	B61001I	B61001M
B61001R	B61001W	B67001H	B83A07A	B83A07B	B83A07C	B83E01C
B83E01D	B83E01E	B85001D	B85008D	B91001A	B91002A	B91002B
B91002C	B91002D	B91002E	B91002F	B91002G	B91002H	B91002I
B91002J	B91002K	B91002L	B95030A	B95061A	B95061F	B95061G
B95077A	B97103E	B97104G	BA1001A	BA1101B	BC1109A	BC1109C
BC1109D	BC1202A	BC1202F	BC1202G	BE2210A	BE2413A	

"PRAGMA ELABORATE (REPORT)" has been added at appropriate points in order to solve the elaboration problems for:

C83030C C86007A

The value used to specify the collection size has been increased from 256 to 324 take alignment into account for:

CD2A83A

CHAPTER 3
PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For each chapter, a command file was generated that loaded and executed every program.

For a point of contact for technical information about this Ada implementation system, see:

Mr. Svend Bodilsen
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 Lyngby
DENMARK

Telephone: + 45 42 87 11 44
Telefax: + 45 42 87 22 17

For a point of contact for sales information about this Ada implementation system, see:

In the U.S.A.:

Mr. Mike Turner
DDC-I, Inc.
9630 North 25th Avenue
Suite #118
Phoenix, Arizona 85021

Mailing address:

P.O. Box 37767
Phoenix, Arizona 85069-7767
Telephone: 602-944-1883
Telefax: 602-944-3253

In the rest of the world:

Mr. Palle Andersson
DDC International A/S
Gl. Lundtoftevej 1B
DK-2800 LYNGBY

Denmark

Telephone: + 45 42 87 11 44

Telefax: + 45 42 87 22 17

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a) Total Number of Applicable Tests	3800	
b) Total Number of Withdrawn Tests	81	
c) Processed Inapplicable Tests	289	
d) Non-Processed I/O Tests	0	
e) Non-Processed Floating-Point Precision Tests	0	
f) Total Number of Inapplicable Tests	289	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

Version 1.11 of the ACVC comprises 4170 tests. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors. The AVF determined that 289 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing. In addition, the modified tests mentioned in section 2.3 were also processed.

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The test suite was loaded onto a VAX-8530 from the magnetic tape. The test suite was then downloaded onto the SUN via Ethernet (using DNICP net software utility).

The tests were compiled, linked and executed on the host/target computer system, as appropriate. The results were captured on

the host/target and transferred to the VAX-8530 computer system using the communications link described above.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. The options invoked explicitly for validation testing during this test were:

-l -a

The options invoked by default for validation testing during this test were:

-x

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. Selected listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is 126 the value for \$MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
\$MAX_IN_LEN	126
\$BIG_ID1	(1..V-1 => 'A', V => '1')
\$BIG_ID2	(1..V-1 => 'A', V => '2')
\$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
\$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
\$BIG_INT_LIT	(1..V-3 => '0') & "298"
\$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
\$BIG_STRING1	"" & (1..V/2 => 'A') & ""
\$BIG_STRING2	"" & (1..V-1-V/2 => 'A') & '1' & ""
\$BLANKS	(1..V-20 => ' ')
\$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
\$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
\$MAX_STRING_LITERAL	"" & (1..V-2 => 'A') & ""

The following table contains the values for the remaining macro parameters.

Macro Parameter	Macro Value
ACC_SIZE	: 32
ALIGNMENT	: 4
COUNT_LAST	: 2_147_483_647
DEFAULT_MEM_SIZE	: 2097152
DEFAULT_STOR_UNIT	: 8
DEFAULT_SYS_NAME	: SUN
DELTA_DOC	: 2#1.0#E-31
ENTRY_ADDRESS	: 16#FF#
ENTRY_ADDRESS1	: 16#FE#
ENTRY_ADDRESS2	: 16#FD#
FIELD_LAST	: 35
FILE_TERMINATOR	: ' '
FIXED_NAME	: NO_SUCH_TYPE
FLOAT_NAME	: NO_SUCH_TYPE
FORM_STRING	: ""
FORM_STRING2	:
"CANNOT_RESTRICT_FILE_CAPACITY"	
GREATER_THAN_DURATION	: 100000.0
GREATER_THAN_DURATION_BASE_LAST	: 200000.0
GREATER_THAN_FLOAT_BASE_LAST	: 16#1.0#E+32
GREATER_THAN_FLOAT_SAFE_LARGE	: 16#5.FFFF_F0#E+31
GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	: 1.0E308
HIGH_PRIORITY	: 32
ILLEGAL_EXTERNAL_FILE_NAME1	: /NODIRECTORY/FILENAME
ILLEGAL_EXTERNAL_FILE_NAME2	: /NODIRECTORY/FILENAME
INAPPROPRIATE_LINE_LENGTH	: -1
INAPPROPRIATE_PAGE_LENGTH	: -1
INCLUDE_PRAGMA1	:
PRAGMA INCLUDE ("A28006D1.TST")	
INCLUDE_PRAGMA2	:
PRAGMA INCLUDE ("B28006E1.TST")	
INTEGER_FIRST	: -2147483648
INTEGER_LAST	: 2147483647
INTEGER_LAST_PLUS_1	: 2147483648
INTERFACE_LANGUAGE	: AS
LESS_THAN_DURATION	: -75000.0
LESS_THAN_DURATION_BASE_FIRST	: -131073.0
LINE_TERMINATOR	: ' '
LOW_PRIORITY	: 1
MACHINE_CODE_STATEMENT	:
AA_INSTR'(AA_EXIT_SUBPRGRM,0,0,0,AA_INSTR_INTG'FIRST,0);	
MACHINE_CODE_TYPE	: AA_INSTR
MANTISSA_DOC	: 31
MAX_DIGITS	: 15

```

MAX_INT                : 2147483647
MAX_INT_PLUS_1         : 2147483648
MIN_INT                : -2147483648
NAME                   : NO_SUCH_TYPE_AVAILABLE
NAME_LIST              : SUN
NAME_SPECIFICATION1    :
/home/sun3/ada/release_46_11/validation/work/X2120A
NAME_SPECIFICATION2    :
/home/sun3/ada/release_46_11/validation/work/X2120B
NAME_SPECIFICATION3    :
/home/sun3/ada/release_46_11/validation/work/X3119A
NEG_BASED_INT          : 16#F000000E#
NEW_MEM_SIZE           : 2097152
NEW_STOR_UNIT          : 8
NEW_SYS_NAME           : SUN
PAGE_TERMINATOR        : ' '
RECORD_DEFINITION      : RECORD
INSTR_NO:INTEGER;ARG0:INTEGER;ARG1:INTEGER;ARG2:INTEGER;ARG3:INTE
GER;ARG4:INTEGER;END RECORD;
RECORD_NAME            : AA_INSTR
TASK_SIZE              : 32
TASK_STORAGE_SIZE      : 1024
TICK                   : 1.0
VARIABLE_ADDRESS       : 16#ffff00#
VARIABLE_ADDRESS1      : 16#ffff20#
VARIABLE_ADDRESS2      : 16#ffff40#
YOUR_PRAGMA            : NOFLOAT

```

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

<u>QUALIFIER</u>	<u>DESCRIPTION</u>
-a	Specifies current sublibrary.
-B	Build standard.
-c <file>	Specifies the configuration file.
-L/-l	Generates list file.
-n	Suppress all run-time checks.
-N <keyword>	Selective suppress of run-time checks.
-o	Optimize.
-O <keyword>	Toggle optimization.
-P	Progress report.
-S/-s	Specifies that the source text is not to be saved in the program library.
-u <unit_number>	Specifies that the compilation unit being compiled is assigned the unit number <unit_number> in the current sublibrary.
-x	Creates a cross reference listing.

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

<u>QUALIFIER</u>	<u>DESCRIPTION</u>
-a	The library used in the link.
-A	Do not include trace back information.
-b	Simulates recompilation of library unit bodies.
-c	Performs consistency check.
-D	Default stack size.
-I	Timer resolution.
-l	Produce a log file.
-L	Produce a log file.
-M	Trace back mode.
-o	Additional object files or object libraries.
-p	Options to native linker.
-P	Default priority.
-R	Do not perform time slice.
-s	Simulates recompilation of specifications.
-S	Time slice.
-T	For maintenance purpose only.
<unit-name>	The name of the main unit.

Default options are: -I (100 ms), -S (10 periods), -P (16), -D (10KBytes)

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

type SHORT_INTEGER is range -32_768 .. 32_767;

type INTEGER is range -2_147_483_648 .. 2_147_483_647;

type FLOAT is digits 6
range -16#0.FFFF_FF#E32 .. 16#0.FFFF_FF#E32;

type LONG_FLOAT is digits 15
range -16#0.FFFF_FFFF_FFFF_F8#E256 ..
16#0.FFFF_FFFF_FFFF_F8#E256;

type DURATION is delta 2#1.0#E-14 range -131_072.0 .. 131_071.0;

end STANDARD;

Appendix

F Appendix F of the Ada Reference Manual

F.1 Introduction

This appendix describes the implementation-dependent characteristics of the DDC-I Sun-3/SunOS V Ada Compiler, as required in the Appendix F frame of the Ada Reference Manual (ANSI/MIL-STD 1815A).

F.2 Implementation-Dependent Pragmas

There is one implementation-defined pragma: `Interface_spelling`, see section 5.6.6.2.

F.3 Implementation-Dependent Attributes

No implementation-dependent attributes are defined.

Appendix

F.4 Package SYSTEM

```
pragma page;  
package SYSTEM is
```

```
    type ADDRESS          is new  INTEGER;  
    subtype PRIORITY      is INTEGER range 1 .. 32;  
    type NAME              is ( SUN );  
    SYSTEM_NAME:          constant NAME := SUN;  
    STORAGE_UNIT:         constant      := 8;  
    MEMORY_SIZE:          constant      := 2048 * 1024;  
    MIN_INT:              constant      := -2_147_483_648;  
    MAX_INT:              constant      := 2_147_483_647;  
    MAX_DIGITS:           constant      := 15;  
    MAX_MANTISSA:         constant      := 31;  
  
    FINE_DELTA:           constant      := 2#1.0#E-31;  
    TICK:                 constant      := 1.0;
```

```
    type interface_language is (C,AS);
```

```
-- Compiler system dependent types:
```

```
    SUBTYPE Integer_16  IS short_integer;  
    SUBTYPE Natural_16  IS Integer_16 RANGE 0..Integer_16'LAST;  
    SUBTYPE Positive_16 IS Integer_16 RANGE 1..Integer_16'LAST;  
  
    SUBTYPE Integer_32  IS integer;  
    SUBTYPE Natural_32  IS Integer_32 RANGE 0..Integer_32'LAST;  
    SUBTYPE Positive_32 IS Integer_32 RANGE 1..Integer_32'LAST;
```

```
end SYSTEM;
```

Appendix

F.5 Representation Clauses

The representation clauses that are accepted are described below. Note that representation specifications can be given on derived types too.

F.5.1 Pragma PACK

Pragma PACK applied on an array type will pack each array element into the smallest number of bits possible, assuming that the component type is a discrete type other than LONG_INTEGER or a fixed point type. Packing of arrays having other kinds of component types have no effect.

When the smallest number of bits needed to hold any value of a type is calculated, the range of the types is extended to include zero.

Pragma PACK applied on a record type will attempt to pack the components not already covered by a representation clause (perhaps none). This packing will begin with the small scalar components and larger components will follow in the order specified in the record. The packing begins at the first storage unit after the components with representation clauses.

The component types in question are the ones defined above for array types.

F.5.2 Length Clauses

Four kinds of length clauses are accepted.

Size specifications:

The size attribute for a type T is accepted in the following cases:

Appendix

- If T is a discrete type then the specified size must be greater than or equal to the number of bits needed to represent a value of the type, and less than or equal to 32. Note that when the number of bits needed to hold any value of the type is calculated, the range is extended to include 0 if necessary, i.e. the range 3..4 cannot be represented in 1 bit, but needs 3 bits.
- If T is a fixed point type, then the specified size must be greater than or equal to the smallest number of bits needed to hold any value of the fixed point type, and less than 32 bits. Note that the Reference Manual permits a representation, where the lower bound and the upper bound is not representable in the type. Thus the type

type FIX is delta 1.0 range -1.0 .. 7.0;

is representable in 3 bits. As for discrete types, the number of bits needed for a fixed point type is calculated using the range of the fixed point type possibly extended to include 0.0.

- If T is a floating point type, an access type or a task type the specified size must be equal to the number of bits used to represent values of the type (floating points: 32 or 64, access types : 32 bits and task types : 32 bits).
- If T is a record type the specified size must be greater than or equal to the minimal number of bits used to represent values of the type per default.
- If T is an array type the size of the array must be static, i.e. known at compile time and the specified size must be equal to the minimal number of bits used to represent values of the type per default.

Furthermore, the size attribute has only effect if the type is part of a composite type.

```
type BYTE is range 0..255;
for BYTE'size use 8;
SIXTEEN : BYTE                -- one word allocated
EIGHT : array(1.4) of BYTE    -- one byte per element
```

Collection size specifications:

Using the STORAGE_SIZE attribute on an access type will set an

Appendix

upper limit on the total size of objects allocated in the collection allocated for the access type. If further allocation is attempted, the exception `STORAGE_ERROR` is raised. The specified storage size must be less than or equal to `INTEGER'LAST`.

Task storage size:

When the `STORAGE_SIZE` attribute is given on a task type, the task stack area will be of the specified size. There is no upper limit on the given size.

Small specifications:

Any value of the `SMALL` attribute less than the specified delta for the fixed point type can be given.

F.5.3 Enumeration Representation Clauses

Enumeration representation clauses may specify representations in the range of `INTEGER'FIRST+1 .. INTEGER'LAST-1`. An enumeration representation clause may be combined with a length clause. If an enumeration representation clause has been given for a type the representational values are considered when the number of bits needed to hold any value of the type is evaluated. Thus the type

```
type ENUM is (A,B,C);  
for ENUM use (1,3,5);
```

needs 3 bits not 2 bits to represent any value of the type.

F.5.4 Record Representation Clauses

When component clauses are applied to a record type the following restrictions and interpretations are imposed :

- All values of the component type must be representable within the specified number of bits in the component clause.
- If the component type is either a discrete type, a fixed point type, an array type with a discrete type other than `LONG_INTEGER`, or a fixed point type as element type, then

Appendix

the component is packed into the specified number of bits (see however the restriction in the paragraph above), and the component may start at any bit boundary.

- If the component type is not one of the types specified in the paragraph above, it must start at a storage unit boundary, a storage unit being 16 bits, and the default size calculated by the compiler must be given as the bit width, i.e. the component must be specified as

component at N range 0 .. 16 * M-1

where N specifies the relative storage unit number (0,1,...) from the beginning of the record, and M the required number of storage units (1,2,...).

- The maximum bit width for components of scalar types is 32.
- A record occupies an integral number of storage units (even though a record may have fields that only define an odd number of bytes)
- A record may take up a maximum of 32 Kbits
- If the component type is an array type with a discrete type other than LONG_INTEGER or a fixed point type as element type, the given bit width must be divisible by the length of the array, i.e. each array element will occupy the same number of bits.

If the record type contains components which are not covered by a component clause, they are allocated consecutively after the component with the value. Allocation of a record component without a component clause is always aligned on a storage unit boundary. Holes created because of component clauses are not otherwise utilized by the compiler.

F.5.4.1 Alignment Clauses

Alignment clauses for records are implemented with the following characteristics :

- If the declaration of the record type is done at the outermost level in a library package, any alignment is accepted.
- If the record declaration is done at a given static level

Appendix

(higher than the outermost library level, i.e. the permanent area), only long word alignments are accepted.

F.6 Implementation-Dependent Names for Implementation-Dependent Components

None defined by the compiler.

F.7 Address Clauses

This section describes the implementation of address clauses and what types of entities may have their address specified by the user.

F.7.1 Objects

Address clauses are supported for scalar and composite objects whose size can be determined at compile time. The address value must be static. The given address is the virtual address.

Appendix

F.8 Unchecked Conversions

Unchecked conversion is only allowed for types where objects have the same "size". The size of an object is interpreted as follows

- for arrays it is the number of storage units occupied by the array elements
- for records it is the size of the fixed part of the record, i.e. excluding any dynamic storage allocated outside the record
- for the other non-structured type, the object size is as described in Chapter 9

For scalar types having a size specification special rules apply. Conversion involving such a type is allowed if the given size matched either the specified size or the object size.

Example

```
type ACC is access INTEGER;
function TO_INT is new UNCHECKED_CONVERSION(ACC, INTEGER);

-- OK

function TO_ACC is new
UNCHECKED_CONVERSION(SHORT_INTEGER, ACC, I);

-- NOT OK

type UNSIGNED is range 0..65535;
for UNSIGNED'SIZE use 16;

function TO_INT is new UNCHECKED_CONVERSION(UNSIGNED, INTEGER);

-- OK

function TO_SHORT is new
UNCHECKED_CONVERSION(UNSIGNED, SHORT_INTEGER);

-- OK

<End example>
```

Appendix

F.9 Input/Output Packages

The implementation supports all requirements of the Ada language and the POSIX standard described in document P1003.5 Draft 4.0/WG15-N45. It is an effective interface to the UNIX file system, and in the case of text I/O, it is also an effective interface to the UNIX standard input, standard output, and standard error streams.

This section describes the functional aspects of the interface to the UNIX file system, including the methods of using the interface to take advantage of the file control facilities provided.

The Ada input-output concept as defined in Chapter 14 of the ARM does not constitute a complete functional specification of the input-output packages. Some aspects of the I/O system are not described at all, with others intentionally left open for implementation. This section describes those sections not covered in the ARM. Please notice that the POSIX standard puts restrictions on some of the aspects not described in Chapter 14 of the ARM.

The UNIX operating system considers all files to be sequences of characters. Files can either be accessed sequentially or randomly. Files are not structured into records, but an access routine can treat a file as a sequence of records if it arranges the record level input-output.

Note that for sequential or text files (Ada files not UNIX external files) RESET on a file in mode OUT_FILE will empty the file. Also, a sequential or text file opened as an OUT_FILE will be emptied.

F.9.1 External Files

An external file is either a UNIX disk file, a UNIX FIFO (named pipe), a UNIX pipe, or any device defined in the UNIX directory. The use of devices such as a tape drive or communication line may require special access permissions or have restrictions. If an inappropriate operation is attempted on a device, the USE_ERROR exception is raised.

External files created within the UNIX file system shall exist after the termination of the program that created it, and will be accessible from other Ada programs. However, pipes and

Appendix

temporary files will not exist after program termination.

Creation of a file with the same name as an existing external file will cause the existing file to be overwritten.

Creation of files with mode `IN_FILE` will cause `USE_ERROR` to be raised.

The name parameter to the input-output routines must be a valid UNIX file name. If the name parameter is empty, then a temporary file is created in the `/usr/tmp` directory. Temporary files are automatically deleted when they are closed

Appendix

F.9.2 File Management

This section provides useful information for performing file management functions within an Ada program.

The only restrictions in performing Sequential and Direct I/O are:

- The maximum size of an object of ELEMENT_TYPE is 2_147_483_647 bits.
- If the size of an object of ELEMENT_TYPE is variable, the maximum size must be determinable at the point of instantiation from the value of the SIZE attribute.

The NAME parameter

The NAME parameter must be a valid UNIX pathname (unless it is the empty string). If any directory in the pathname is inaccessible, a USE_ERROR or a NAME_ERROR is raised.

The UNIX names "stdin", "stdout", and "stderr" can be used with TEXT_IO.OPEN. No physical opening of the external file is performed and the internal Ada file will be associated with the already open external file. These names have no significance for other I/O packages.

Temporary files (NAME = null string) are created using tmpname(3) and are deleted when CLOSED. Abnormal program termination may leave temporary files in existence. The name function will return the full name of a temporary file when it exists.

The FORM parameter

The Form parameter, as described below, is applicable to DIRECT_IO, SEQUENTIAL_IO and TEXT_IO operations. The value of the Form parameter for Ada I/O shall be a character string. The value of the character string shall be a series of fields separated by commas. Each field shall consist of optional separators, followed by a field name identifier, followed by optional separators, followed by "=>", followed by optional separators, followed by a field value, followed by optional separators. The allowed values for the field names and the corresponding field values are described below. All field names

Appendix

and field values are case-insensitive.

The following BNF describes the syntax of the FORM parameter:

```
form                ::= [field {, field}]*

fields               ::= rights | append | blocking |
                       terminal_input | fifo |
                       posix_file_descriptor

rights               ::= OWNER | GROUP | WORLD =>
                       access {,access_underscor}

access               ::= READ | WRITE | EXECUTE | NONE

access_underscor     ::= _READ | _WRITE | _EXECUTE | _NONE

append               ::= APPEND => YES | NO

blocking             ::= BLOCKING => TASKS | PROGRAM

terminal_input       ::= TERMINAL_INPUT => LINES | CHARACTERS

fifo                 ::= FIFO => YES | NO

posix_file_descriptor ::= POSIX_FILE_DESCRIPTOR => 2
```

The FORM parameter is used to control the following :

- File ownership:

Access rights to a file is controlled by the following field names "OWNER", "GROUP" and "WORLD". The field values are "READ", "WRITE", "EXECUTE" and "NONE" or any combination of the previously listed values separated by underscores. The access rights field names are applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. The default value is OWNER => READ_WRITE, GROUP => READ_WRITE and WORLD => READ_WRITE. The actual access rights on a created file will be the default value subtracted the value of the environment variable **umask**.

Example

To make a file readable and writable by the owner only, the Form parameter should look something like this:

Appendix

```
"Owner =>read_write, World=> none, Group=>none"
```

If one or more of the field names are missing the default value is used. The permission field is evaluated in left-to-right order. An ambiguity may arise with a Form parameter of the following:

```
"Owner=>Read_Execute_None_Write_Read"
```

Appendix

In this instance, using the left-to-right evaluation order, the "None" field will essentially reset the permissions to none and this example would have the access rights WRITE and READ.

- Appending to a file:

Appending to a file is achieved by using the field name "APPEND" and one of the two field values "YES" or "NO". The default value is "NO". "Append" is allowed with both TEXT_IO and SEQUENTIAL_IO. The effect of appending to a file is that all output to that file is written to the end of the named external file. This field may only be used with the "OPEN" operation, using the field name "APPEND" in connection with a "CREATE" operation shall raise USE_ERROR. Furthermore, a USE_ERROR is raised if the specified file is a terminal device or another device.

Example

To append to a file, one would write:

```
"Append => Yes"
```

- Blocking vs. non-blocking I/O:

The blocking field name is "Blocking" and the field values are "TASKS" and "PROGRAM". The default value is "PROGRAM". "Blocking=>Tasks" causes the calling task, but no others, to wait for the completion of an I/O operation. "Blocking=>program" causes the all tasks within the program to wait for the completion of the I/O operation. The blocking mechanism is applicable to TEXT_IO, DIRECT_IO and SEQUENTIAL_IO. UNIX does not allow the support of "BLOCKING=>TASKS" currently.

- How characters are read from the keyboard:

The field name is "TERMINAL_INPUT" and the field value is either "LINES" or "CHARACTERS". The effect of the field value "Terminal_input => Characters" is that characters are read in a noncanonical fashion with Minimum_count=1, meaning one character at a time and Time=0.0 corresponding to that the read operation is not satisfied until Minimum_Count characters are received. If the field value "LINES" is used the characters are read one line at a time

Appendix

in canonical mode. The default value is Lines. "TERMINAL_INPUT" has no effect if the specified file is not already open or if the file is not open on a terminal. It is permitted for the same terminal device to be opened for input in both modes as separate Ada file objects. In this case, no user input characters shall be read from the input device without an explicit input operation on one of the file objects. The "TERMINAL_INPUT" mechanism is only applicable to TEXT_IO.

- Creation of FIFO files:

The field name is "Fifo" and the field value is either "YES" or "NO". "FIFO => YES" means that the file shall be a named FIFO file. The default value is "No".

For use with TEXT_I/O, the "Fifo" field is only allowed with the Create operation. If used in connection with an open operation an USE_ERROR is raised.

For SEQUENTIAL_IO, the FIFO mechanism is applicable for both the Create and Open operation.

In connection with SEQUENTIAL_IO, an additional field name "O_NDELAY" is used. The field values allowed for "O_NDELAY" are "YES" and "NO". Default is "NO". The "O_NDELAY" field name is provided to allow waiting or immediate return. If, for example, the following form parameter is given:

"Fifo=>Yes, O_Ndelay=>Yes"

then waiting is performed until completion of the operation. The "O_Ndelay" field name only has meaning in connection with the FIFO facility and is otherwise ignored.

- Access to Open POSIX files:

The field name is "POSIX_File_Descriptor". The field value is the character string "2" which denotes the stderr file. Any other field value will result in USE_ERROR being raised. The Name parameter provides the value which will be returned by subsequent usage of the Name function. The operation does not change the state of the file. During the period that the Ada file is open, the result of any file operations on the file descriptor are undefined. Note that this is a method to make stderr accessible from an Ada

Appendix

program.

File Access

The following guidelines should be observed when performing file I/O operations:

- At a given instant, any number of files in an Ada program can be associated with corresponding external files.
- When sharing files between programs, it is the responsibility of the programmer to determine the effects of sharing files.
- The RESET and OPEN operations to files with mode OUT_FILE will empty the contents of the file in SEQUENTIAL_IO and TEXT_IO.
- Files can be interchanged between SEQUENTIAL_IO and DIRECT_IO without any special operations if the files are of the same object type.

Appendix

F.9.3 Buffering

The Ada I/O system provides buffering in addition to the buffering provided by UNIX. The Ada TEXT_IO packages will flush all output to the operating system under the following circumstances:

1. The device is a terminal device and an end of line, end of page, or end of file has occurred.
2. The device is a terminal device and the same Ada program makes an Ada TEXT_IO input request or another file object representing the same device.

F.9.4 Package IO_EXCEPTIONS

The specification of package IO_EXCEPTIONS:

package IO_EXCEPTIONS is

-- The order of the following declarations must NOT be changed:

```
STATUS_ERROR      : exception;
MODE_ERROR        : exception;
NAME_ERROR        : exception;
USE_ERROR         : exception;
DEVICE_ERROR      : exception;
END_ERROR         : exception;
DATA_ERROR        : exception;
LAYOUT_ERROR      : exception;
```

end IO_EXCEPTIONS;

Appendix

F.9.5 Package TEXT_IO

The specification of package TEXT_IO:

with BASIC_IO_TYPES;

with IO_EXCEPTIONS;

package TEXT_IO is

 type FILE_TYPE is limited private;

 type FILE_MODE is (IN_FILE, OUT_FILE);

 type COUNT is range 0 .. INTEGER'LAST;

 subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

 UNBOUNDED: constant COUNT:= 0; -- line and page length

 -- max. size of an integer output field 2#....#

 subtype FIELD is INTEGER range 0 .. 65;

 subtype NUMBER_BASE is INTEGER range 2 .. 16;

 type TYPE_SET is (LOWER_CASE, UPPER_CASE);

 -- File Management

```
procedure CREATE (FILE      : in out FILE_TYPE;
                  MODE      : in FILE_MODE :=OUT_FILE;
                  NAME      : in STRING   :="";
                  FORM      : in STRING   :=""
                  );
```

```
procedure OPEN (  FILE      : in out FILE_TYPE;
                 MODE      : in FILE_MODE;
                 NAME      : in STRING;
                 FORM      : in STRING   :=""
                 );
```

```
procedure CLOSE (FILE : in out FILE_TYPE)
```

```
procedure DELETE      (FILE : in out FILE_TYPE);
```

```
procedure RESET (FILE : in out FILE_TYPE;  MODE: in
                  FILE_MODE);
```

```
procedure RESET (FILE : in out FILE_TYPE);
```

```
function MODE      (FILE : in FILE_TYPE) return FILE_MODE;
```

```
function NAME      (FILE : in FILE_TYPE) return STRING;
```

```
function FORM      (FILE : in FILE_TYPE) return STRING;
```

```
function IS_OPEN      (FILE : in FILE_TYPE return BOOLEAN;
```

Appendix

```
-- control of default input and output files

procedure SET_INPUT      (FILE : in FILE_TYPE);
procedure SET_OUTPUT     (FILE : in FILE_TYPE);

function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;

function CURRENT_INPUT  return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;

-- specification of line and page lengths

procedure SET_LINE_LENGTH (FILE : in FILE_TYPE; TO : in
                           COUNT);
procedure SET_LINE_LENGTH (TO : in COUNT);

procedure SET_PAGE_LENGTH (FILE : in FILE_TYPE; TO : in
                           COUNT);
procedure SET_PAGE_LENGTH (TO : in COUNT);

function LINE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function LINE_LENGTH return COUNT;

function PAGE_LENGTH (FILE : in FILE_TYPE) return COUNT;
function PAGE_LENGTH return COUNT;

-- Column, Line, and Page Control

procedure NEW_LINE (FILE : in FILE_TYPE;
                   SPACING : in POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING : in POSITIVE_COUNT := 1);

procedure SKIP_LINE (FILE : in FILE_TYPE;
                   SPACING : in POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING : in POSITIVE_COUNT := 1);

function END_OF_LINE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_LINE return BOOLEAN;

procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE;

procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE;

function END_OF_PAGE (FILE : in FILE_TYPE) return BOOLEAN;
function END_OF_PAGE return BOOLEAN;
```

Appendix

```

function END_OF_FILE      (FILE : in FILE_TYPE) return BOOLEAN;
function  END_OF_FILE      return BOOLEAN;

procedure SET_COL          (FILE : in FILE_TYPE; TO : in
                           POSITIVE_COUNT);
procedure SET_COL          (TO : in POSITIVE_COUNT);

procedure SET_LINE         (FILE : in FILE_TYPE; TO : in
                           POSITIVE_COUNT);
procedure SET_LINE         (TO : in POSITIVE_COUNT);

function COL               (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function COL               return POSITIVE_COUNT;

function LINE              (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function LINE              return POSITIVE_COUNT;

function PAGE              (FILE : in FILE_TYPE) return POSITIVE_COUNT;
function PAGE              return POSITIVE_COUNT;

-- Character Input-Output

procedure GET              (FILE : in FILE_TYPE; ITEM : out CHARACTER);
procedure GET              (ITEM : out CHARACTER);
procedure PUT              (FILE : in FILE_TYPE; ITEM : in CHARACTER);
procedure PUT              (ITEM : in CHARACTER);

-- String Input-Output

procedure GET              (FILE : in FILE_TYPE; ITEM : out STRING);
procedure GET              (ITEM : out STRING);
procedure PUT              (FILE : in FILE_TYPE; ITEM : in STRING);
procedure PUT              (ITEM : in STRING);

procedure GET_LINE         (FILE : in FILE_TYPE;
                           ITEM : out STRING;
                           LAST : out NATURAL);

procedure GET_LINE         (ITEM : out STRING;      LAST : out
                           NATURAL);
procedure PUT_LINE         (FILE : in FILE_TYPE;   ITEM : in
                           STRING);
procedure PUT_LINE         (ITEM : in STRING);
-- Generic Package for Input-Output of Integer Types

generic
    type NUM is range <>;
package INTEGER_IO is

```

Appendix

```

DEFAULT_WIDTH : FIELD           := NUM'WIDTH;
DEFAULT_BASE  : NUMBER_BASE     := 10;

procedure GET  (FILE   : in FILE_TYPE;
               ITEM    : out NUM;
               WIDTH   : in FIELD := 0);

procedure GET  (ITEM    : out NUM;
               WIDTH   : in FIELD := 0);

procedure PUT  (FILE   : in FILE_TYPE;
               ITEM    : in NUM;
               WIDTH   : in FIELD := DEFAULT_WIDTH;
               BASE    : in NUMBER_BASE:= DEFAULT_BASE);

procedure PUT  (ITEM    : in NUM;
               WIDTH   : in FIELD := DEFAULT_WIDTH;
               BASE    : in NUMBER_BASE:= DEFAULT_BASE);

procedure GET  (FROM    : in STRING;
               ITEM     : out NUM;
               LAST     : out POSITIVE);

procedure PUT  (TO      : out STRING;
               ITEM     : in NUM;
               BASE     : in NUMBER_BASE:= DEFAULT_BASE);

end INTEGER_IO;

-- Generic Packages for Input-Output of Real Types

generic
  type NUM is digits <>;
package FLOAT_IO; is

  DEFAULT_FORE : FIELD := 2;
  DEFAULT_AFT  : FIELD := NUM'DIGITS - 1;
  DEFAULT_EXP  : FIELD := 3;

  procedure GET  (FILE   : in FILE_TYPE;
                 ITEM    : out NUM;
                 WIDTH   : in FIELD := 0);

  procedure GET  (ITEM    : out NUM;
                 WIDTH   : in FIELD := 0);

  procedure PUT  (FILE   : in FILE_TYPE;
                 ITEM    : in NUM;
                 FORE     : in FIELD := DEFAULT_FORE;

```

Appendix

```

                                AFT      : in FIELD      := DEFAULT_AFT;
                                EXP      : in FIELD      := DEFAULT_EXP);
procedure PUT (ITEM           : in NUM;
              FORE           : in FIELD      := DEFAULT_FORE;
              AFT           : in FIELD      := DEFAULT_AFT;
              EXP           : in FIELD      := DEFAULT_EXP);
procedure GET (FROM         : in STRING;
              ITEM         : out NUM;
              LAST         : out POSITIVE);
procedure PUT (TO           : out STRING;
              ITEM         : in NUM;
              AFT          : in FIELD      := DEFAULT_AFT;
              EXP          : in FIELD      := DEFAULT_EXP);

end FLOAT_IO;
generic
  type NUM is delta <>;
package FIXED_IO is

  DEFAULT_FORE : FIELD := NUM'FORE;
  DEFAULT_AFT  : FIELD := NUM'AFT;
  DEFAULT_EXP  : FIELD := 0;

  procedure GET (FILE       : in FILE_TYPE;
                ITEM       : out NUM;
                WIDTH      : in FIELD      := 0);
  procedure GET (ITEM      : out NUM;
                WIDTH      : in FIELD      := 0);

  procedure PUT (FILE       : in FILE_TYPE;
                ITEM       : in NUM;
                FORE       : in FIELD      := DEFAULT_FORE;
                AFT       : in FIELD      := DEFAULT_AFT;
                EXP       : in FIELD      := DEFAULT_EXP);
  procedure PUT (ITEM      : in NUM;
                FORE       : in FIELD      := DEFAULT_FORE;
                AFT       : in FIELD      := DEFAULT_AFT;
                EXP       : in FIELD      := DEFAULT_EXP);

  procedure GET (FROM      : in STRING;
                ITEM      : out NUM;
                LAST      : out POSITIVE);

  procedure PUT (TO        : out STRING;
                ITEM      : in NUM;
                AFT       : in FIELD      := DEFAULT_AFT;
                EXP       : in FIELD      := DEFAULT_EXP);

end FIXED_IO;

```

Appendix

```
-- Generic Package for Input-Output of Enumeration Types

generic
  type ENUM is (<>);
package ENUMERATION_IO is

  DEFAULT_WIDTH    : FIELD          := 0;
  DEFAULT_SETTING  : TYPE_SET       := UPPER_CASE;

  procedure GET (FILE      : in FILE_TYPE; ITEM : out ENUM);
  procedure GET (ITEM      : out ENUM);

  procedure PUT (FILE      : FILE_TYPE;
                ITEM       : in ENUM;
                WIDTH      : in FIELD   := DEFAULT_WIDTH;
                SET        : in TYPE_SET := DEFAULT_SETTING);

  procedure PUT (ITEM      : in ENUM;
                WIDTH      : in FIELD   := DEFAULT_WIDTH;
                SET        : in TYPE_SET := DEFAULT_SETTING);

  procedure GET (FROM      : in  STRING;
                ITEM       : out ENUM;
                LAST       : out POSITIVE);

  procedure PUT (TO        : out STRING;
                ITEM       : in  ENUM;
                SET        : in  TYPE_SET := DEFAULT_SETTING);

end ENUMERATION_IO;

-- Exceptions

STATUS_ERROR    : exception renames
                  IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR      : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR      : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR       : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR    : exception renames
                  IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR       : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR      : exception renames IO_EXCEPTIONS.DATA_ERROR;
LAYOUT_ERROR    : exception renames
                  IO_EXCEPTIONS.LAYOUT_ERROR;

private

  type FILE_BLOCK_TYPE is new BASIC_IO_TYPES.FILE_TYPE;
```

Appendix

```
type FILE_OBJECT_TYPE is
  record
    IS_OPEN      : BOOLEAN          := FALSE;
    FILE_BLOCK   : FILE_BLOCK_TYPE;
  end record;

type FILE_TYPE is access FILE_OBJECT_TYPE;
end TEXT_IO;
```

Appendix

F.9.6 Package SEQUENTIAL_IO

```
-- Source code for SEQUENTIAL_IO
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic

    type ELEMENT_TYPE is private;

package SEQUENTIAL_IO is

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE);

-- File management

    procedure CREATE(FILE : in out FILE_TYPE;
        MODE      : in    FILE_MODE      := OUT_FILE;
        NAME      : in    STRING         := "";
        FORM      : in    STRING         := "");

    procedure OPEN (FILE      : in out FILE_TYPE;
        MODE      : in    FILE_MODE;
        NAME      : in    STRING;
        FORM      : in    STRING := "");

    procedure CLOSE (FILE      : in out FILE_TYPE);

    procedure DELETE(FILE      : in out FILE_TYPE);

    procedure RESET (FILE      : in out FILE_TYPE; MODE : in
FILE_MODE);

    procedure RESET (FILE      : in out FILE_TYPE);

    function MODE      (FILE      : in FILE_TYPE) return FILE_MODE;

    function NAME      (FILE      : in FILE_TYPE) return STRING;

    function FORM      (FILE      : in FILE_TYPE) return STRING;

    function IS_OPEN(FILE      : in FILE_TYPE) return BOOLEAN;

-- input and output operations

    procedure READ (FILE      : in FILE_TYPE;
```


Appendix

```
ITEM : out ELEMENT_TYPE);

procedure WRITE (FILE : in FILE_TYPE;
                ITEM : in ELEMENT_TYPE);

function END_OF_FILE
    (FILE : in FILE_TYPE) return BOOLEAN;

-- exceptions

STATUS_ERROR : exception renames
IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR : exception renames
IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames IO_EXCEPTIONS.DATA_ERROR;

private

type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end SEQUENTIAL_IO;
```

Appendix

F.9.7 Package DIRECT_IO

```
with BASIC_IO_TYPES;
with IO_EXCEPTIONS;

generic

    type ELEMENT_TYPE is private;

package DIRECT_IO is

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

    type COUNT is range 0..INTEGER'LAST;

    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;

    -- File management

    procedure CREATE(FILE      : in out FILE_TYPE;
                     MODE      : in      FILE_MODE:= INOUT_FILE;
                     NAME      : in      STRING  := "";
                     FORM      : in      STRING  := "");

    procedure OPEN  (FILE      : in out FILE_TYPE;
                     MODE      : in      FILE_MODE;
                     NAME      : in      STRING;
                     FORM      : in      STRING := "");

    procedure CLOSE (FILE      : in out FILE_TYPE);

    procedure DELETE(FILE      : in out FILE_TYPE);

    procedure RESET (FILE      : in out FILE_TYPE;
                     MODE      : in      FILE_MODE);

    procedure RESET (FILE      : in out FILE_TYPE);

    function MODE    (FILE      : in      FILE_TYPE) return FILE_MODE;

    function NAME    (FILE      : in      FILE_TYPE) return STRING;

    function FORM    (FILE      : in      FILE_TYPE) return STRING;

    function IS_OPEN(FILE      : in      FILE_TYPE) return BOOLEAN;

    -- input and output operations
```

Appendix

```

procedure READ  (FILE      : in      FILE_TYPE;
                 ITEM      : out     ELEMENT_TYPE;
                 FROM      : in      POSITIVE_COUNT);
procedure READ  (FILE      : in      FILE_TYPE;
                 ITEM      : out     ELEMENT_TYPE);

procedure WRITE (FILE      : in      FILE_TYPE;
                 ITEM      : in      ELEMENT_TYPE;
                 TO        : in      POSITIVE_COUNT);
procedure WRITE (FILE      : in      FILE_TYPE;
                 ITEM      : in      ELEMENT_TYPE);

procedure SET_INDEX
      (FILE      : in FILE_TYPE;
       TO        : in POSITIVE_COUNT);

function INDEX  (FILE      : in      FILE_TYPE) return
POSITIVE_COUNT;

function SIZE   (FILE      : in FILE_TYPE) return COUNT;

function END_OF_FILE
      (FILE      : in FILE_TYPE) return BOOLEAN;

-- exceptions

STATUS_ERROR    : exception renames
                  IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR      : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR      : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR       : exception renames IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR    : exception renames
IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR       : exception renames IO_EXCEPTIONS.END_ERROR;
DATA_ERROR      : exception renames IO_EXCEPTIONS.DATA_ERROR;

private
  type FILE_TYPE is new BASIC_IO_TYPES.FILE_TYPE;

end DIRECT_IO;

```

Appendix

F.9.8 Package TERMINAL

The specification of package TERMINAL:

```
with COMMON_DEFS;  
use COMMON_DEFS;
```

package TERMINAL is

```
    procedure SET_CURSOR(ROW, COL          : in      INTEGER);  
  
    procedure IN_CHARACTER(CH      :      out CHARACTER);  
  
    procedure IN_INTEGER  (I      :      out INTEGER);  
  
    procedure IN_LINE     (T      :      out TERMINAL_LINE);  
  
    procedure OUT_CHARACTER(CH : in      CHARACTER);  
  
    procedure OUT_INTEGER  (I      : in      INTEGER);  
  
    procedure OUT_INTEGER_F(I, W          : in      INTEGER);  
  
    procedure OUT_LINE     (L      : in      STRING);  
  
    procedure OUT_STRING   (S      : in      STRING);  
  
    procedure OUT_NL;  
  
    procedure OUT_FF;  
  
    procedure FLUSH;  
  
    procedure OPEN_LOG_FILE(FILE_NAME      : in      STRING);  
  
    procedure CLOSE_LOG_FILE;
```

end TERMINAL;